Analyzing the co-evolution of comments and source code

Fluri, B ; Würsch, M ; Giger, E ; Gall, H C

Abstract: Source code comments are a valuable instrument to preserve design decisions and to communicate the intent of the code to programmers and maintainers. Nevertheless, commenting source code and keeping them up-to-date is often neglected for reasons of time or programmer's obliviousness. In this paper, we investigate the question whether developers comment their code and to which extent they add comments or adapt them when they evolve the code. We present an approach to associate comments with source code entities to track their co-evolution over multiple versions. A set of heuristics are used to decide whether a comment is associated to its preceding or its succeeding source code entity. We analyzed the co-evolution of code and comments in eight different open source and closed source software systems. We found with statistical significance that (1) the relative amount of comments and source code grows at about the same rate; (2) the type of a source code entity, such as a method declaration or an if-statement, has a significant influence on whether or not it gets commented; (3) in six out of the eight systems, code and comments co-evolve in 90 percent of the cases; and (4) surprisingly, API changes and comments do not co-evolve but they are re-documented in a later revision. As a result, our approach enables a quantitative assessment of the commenting process in a software system. We can, therefore, leverage the results to provide feedback during development to increase the awareness when to add comments or when to adapt comments because of source code changes.

DOI: https://doi.org/10.1007/s11219-009-9075-x

Software Qual J (2009) 17:367–394
DOI 10.1007/s11219-009-9075-x

Analyzing the co-evolution of comments and source code

Beat Fluri · Michael Würsch · Emanuel Giger · Harald C. Gall

Published online: 26 March 2009

Abstract Source code comments are a valuable instrument to preserve design decisions and to communicate the intent of the code to programmers and maintainers. Nevertheless, commenting source code and keeping comments up-to-date is often neglected for reasons of time or programmers obliviousness. In this paper, we investigate the question whether developers comment their code and to what extent they add comments or adapt them when they evolve the code. We present an approach to associate comments with source code entities to track their co-evolution over multiple versions. A set of heuristics are used to decide whether a comment is associated with its preceding or its succeeding source code entity. We analyzed the co-evolution of code and comments in eight different open source and closed source software systems. We found with statistical significance that (1) the relative amount of comments and source code grows at about the same rate; (2) the type of a source code entity, such as a method declaration or an if-statement, has a significant influence on whether or not it gets commented; (3) in six out of the eight systems, code and comments co-evolve in 90% of the cases; and (4) surprisingly, API changes and comments do not co-evolve but they are re-documented in a later revision. As a result, our approach enables a quantitative assessment of the commenting process in a software system. We can, therefore, leverage the results to provide feedback during development to increase the awareness of when to add comments or when to adapt comments because of source code changes.

Keywords Software evolution analysis · Software repositories · Source code changes · Comment changes · Comment quality · Software maintenance

1 Introduction

Documenting software is painful, especially when time is scarce and release deadlines are putting serious pressure on development teams, making it necessary to prioritize tasks.

B. Fluri (✉) · M. Würsch · E. Giger · H. C. Gall
Department of Informatics, University of Zurich, Zurich, Switzerland
e-mail: fluri@ifi.uzh.ch

Under these circumstances feature implementation and bug fixing are top priority because customers usually pay for functionality in the first place and complain about non-functional requirements later on, when the impact of possible deficiencies in maintainability becomes apparent. The task of writing comments is often neglected by developers or given to staff members who are less familiar with the system, although every developer knows the value of good comments (Vanter 2002).

Reading code is a fundamental task during software engineering (Goldberg 1987)—code is read more often than it is written. Comments allow one to understand the code faster and deeper as well as to improve its readability (Spinellis 2006; Tenny 1988). They are crucial to sustaining software maintainability and aid in reverse engineering, for example, when applying the *Read All the Code in One Hour* reengineering pattern (Demeyer et al. 2003). Elshoff and Marcotty (1982) stated that comments as well as the structure of the source code aid in program understanding and therefore reduce maintenance costs. Their finding was confirmed by the studies of Tenny (1988). Nonetheless, the example of Lakhotia (1993) showed that sometimes programmers do not care that someone else might want to understand the source code.

To understand whether the comments are a reason for decreasing maintainability in software projects, we study various productive software systems and address the following research questions in this paper:

1. Is the growth factor the same for source code and comments, meaning that about the same relative amount of code and comments is added over time? During the life cycle of a system, the API becomes more stable, most parts of the implementation have undergone several reviews, and re-documentation during maintenance takes places. We expect that the growth factor of code and comments approximate each other over time and keeps the ratio of commented source code stable.
2. Does the type of the source code entity have an influence on whether it gets commented and which source code entities are commented the most? The answer indicates whether developers are aware that commenting declaration parts and scope (and in numerous other places) increases readability and makes programs more comprehensible and therefore easier to maintain in the long-term.
3. Are comments adapted when source code is changed (i.e., are comments kept up-to-date) and when does the adaptation take place—while changing the source code or in a later revision? By answering this question, we can draw conclusions on whether re-documentation is a integral part in the software engineering process, even though programmers often neglect to adapt documentation to source code changes immediately.

To answer these questions we developed an approach to map comments to source code entities and to track co-changes of source code and comments over the history of a software system. We use the heuristics that the proximity between source code and comments indicate an association, and that comments describe the source code to which they are associated.

To track co-changes we leverage data provided by our Evolizer and ChangeDistiller (Fluri et al. 2007). Both tools are plugins for the Eclipse[1] IDE (des Rivières and Wiegand 2004). Evolizer provides facilities to extract historical information from version control repositories of Java software projects and to store them in a database. ChangeDistiller uses this information to extract fine-grained source code changes between subsequent

[1] http://www.eclipse.org.

revisions of Java classes. The abstraction used by ChangeDistiller focuses on changes in body and declaration parts of attributes, classes, as well as methods, and stops at the statement level. For instance, with ChangeDistiller we can extract that a method invocation statement was moved inside the else-part of an if-statement or that a parameter was added to a method declaration.

When the process of associating comments to source code and extracting co-changes between them is completed we can answer questions like *"What is the most commented source code entity in a method body?"*—e.g., *"it is the if-statement,"* or *"When was the comment associated to a particular if-statement adapted to a condition change?"*—e.g., *"three revisions after the if-condition changed."*

For each research question we explain its rationale, define corresponding hypotheses, and conduct an empirical study with eight software systems. These systems consist of three major components of Eclipse, one commercial system, and four open-source systems from different domains. Based on the results of the studies we statistically show:

1. The growth factor of source code and comments are equal. This does not mean that every new line of code gets commented—in half of the investigated systems newly added code gets barely commented.
2. The type of source code entity highly correlates whether the entity is commented or not and there is also a partial order in the likeliness of whether a certain entity gets commented. For example, if-statements are commented more often than simple statements.
3. Over 50% of the comment changes are related to source code changes. For six of the investigated systems over 90% of these co-changes are applied in the same revision.

The contributions of the this paper are (1) an approach to map comments to source code entities, (2) an approach to track co-changes of source code and comments over the history of a software system, and (3) an empirical study on the commenting behavior in software systems. We also report on the experiences we have had when we applied our approach in industrial projects and provide a discussion on further applications of our work in terms of software quality analysis.

The remainder of this paper is structured as follows. In Sect. 2 we introduce Evolizer as well as ChangeDistiller and illustrate how source code changes from a version control repository are obtained. In Sect. 3 we present our approach to map comments to source code entities and to track co-changes. This approach is then applied to eight software systems and we discuss the results in Sect. 4. The interpretation in terms of software quality of the results is discussed in Sect. 5, including threats to validity to our approach. Finally, related work and conclusions are discussed in Sects. 6 and 7.

2 Overview of Evolizer and ChangeDistiller

To track co-changes we leverage data provided by our Evolizer and ChangeDistiller. We briefly describe them in this section.

Evolizer basically stems from the idea of having a Release History Database (RHDB) (Fischer et al. 2003) that integrates information originating from various repositories, such as CVS and Bugzilla, in a single database. In particular, Evolizer is a set of Eclipse plugins and comparable with *Kenyon* of Bevan et al. (2005) or *eROSE* of Zimmermann et al. (2005).

When importing a version control repository, Evolizer parses the log output of the repository, stores all information provided by the log output, i.e., file name, revision

number, author, commit message, commit date, etc., along with the complete file revision content in our EVOLIZER RHDB. Through interfaces the EVOLIZER provides access to the RHDB.

CHANGEDISTILLER is an implementation of our *change distilling* algorithm (presented in full detail in Fluri et al. 2007) and is also integrated into the Eclipse IDE as a plugin. The aim of CHANGEDISTILLER is to extract fine-grained source code changes applied on subsequent revisions of Java classes which are fetched from the RHDB.

The change distilling algorithm is a tree differencing algorithm customized to be applicable to pairs of abstract syntax trees (AST). For that, the algorithm first finds a matching set between the nodes of the two ASTs. Finding a match between two AST nodes is based on string similarity measures for leaves and tree similarity measures for subtrees.

Second, the algorithm generates an edit script, i.e., a set of atomic changes, that transforms one AST into the other. An atomic change is one of the basic tree edit operations *insert, delete, move*, or *update* applied to an AST node. After generating the edit script, each operation is assigned to a *change type* according to our *taxonomy of source code changes* (Fluri and Gall 2006). For instance, the tree edit operation for the change type *statement parent change* is the move operation of a statement. That means, a statement is moved to a particular position in the method body. We have defined over 35 different change types on body and declaration parts of attributes, classes, and fields. The most fine-grained level of the taxonomy is the statement level.

Leveraging the information provided by ASTs permits us to get exact information about a source code change. In addition to the information that a particular source code entity has changed, tree edit operations also provide information about the location of the change. For instance, we can tell that the method invocation statement `foo.bar()` was moved from the then-part to the else-part of the if-statement that has the condition `foo == null`.

3 Data extraction and collection

To answer our three research questions, we extract and collect data from three different sources. Counting the number of lines of non-commented source code and lines of comments is straightforward and not discussed in-depth. In this section we present our approach to mapping comments to source code entities and to tracking co-changes among them.

Figure 1 gives an overview of the mapping, change detection, and co-change tracking process:

1. The source code of all revisions of a particular Java class is fetched from the EVOLIZER RHDB. Before using these revisions as input for CHANGEDISTILLER, we establish a mapping between comments and source code entities for each revision.
2. For each pair of subsequent revisions, we extract the change types of both the source code entities and the comments with our CHANGEDISTILLER. The change types are stored in the EVOLIZER RHDB.
3. When this process is completed, a fully-fledged change history is available for each class, allowing us to relate comment to source code changes and make a variety of observations, ranging from, e.g., *"The most commented source code entity is..."* to more sophisticated ones such as *"The comment associated with a particular if-statement in method*`bar()` *was adapted three revisions after the condition of the if-statement had been updated."* By aggregating these observations we can analyse the process of adapting comments to source code changes of a software system.

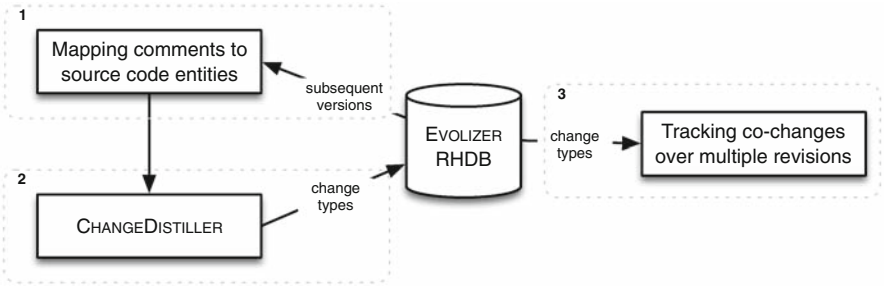


Fig. 1 Overview on the change detection and tracking process

3.1 Mapping comments to source code entities

In programming languages, it is seldom straightforward to track relations between comments and source code entities automatically. Block and line comments cannot be assigned confidently to a particular adjacent entity by using purely syntactical rules. Because of that, Kaelbling (1988) proposed to remove line and block comments from programming languages and to introduce *scoped comments* instead. In today's programming languages, we still have to deal with line and block comments and, consequently, we have to establish a mapping by applying a set of heuristics.

We treat consecutive line comments as a syntactical alternative to block comments and merge them before we establish a mapping between source code entities and comments. Furthermore, we filter dead source code with a regular expression matcher that targets simple source code indicators such as patterns of parentheses, brackets, semicolons, and the like. This approach is lightweight but does not have the power of a full-fledged parser. Still we are able to filter the majority of commented source code with acceptable precision (0.73) and recall (0.65) rates, as explained in detail in Sect. 4.2.1. We do not apply the matcher on API comments (e.g., Javadoc) because they often contain code or code-like structures, such as source code snippets, giving an example of how a class or method is used properly. Another example are (semi-)formal specifications that define pre and postconditions. But we filter empty API comments, since IDEs may construct them when a developer adds a new class, field, or method. Empty API comments are similar to:

```
/**
 *
 */
```

For each comment, we form triples of {preceding source code entity, comment, succeeding source code entity}. We associate a comment with at most one code element. While it is straightforward to associate a Javadoc comment to its following source code entity by using the information contained in the abstract syntax tree generated by the parser of Eclipse, block and line comments within a method's body can either belong to their preceding or succeeding source code entity. To find out whether a comment is associated with its preceding or succeeding source code entity, we apply the following set of heuristics on every triple:

- *Comment on the same line*. Comments and source code entities located on the same line are often associated. These kinds of comments clarify the meaning of the preceding source code entity, as shown in the following example:

```
int i = 0; // Counter for while loop
```

- *Comment on an adjacent line*. Comments are normally in direct proximity of the corresponding source code entity. In the example below, each of the surrounding statements must be considered to be associated:

```
foo();
/* If foo() did not succeed,
   then calling bar() will
   raise an exception. */
bar();
```

- *Comment describes source code*. Each word appearing in the comment as well as in the source code entity is an indication that the comment belongs to the source code entity. We use a token-based measure (see Sect. 3.2 for details) to determine the similarity between comment and source code. We follow the heuristic that comments often pick up identifiers, e.g., variable names, found in the code which they are describing. To separate tokens in comments and source code entities we use non-alphanumeric characters as delimiters. Concerning the example above, both method invocations, `foo()` and `bar()`, can be associated with the comment.

For both, the preceding and the succeeding source code entity, we compute a ranking based on these heuristics. We map the higher ranked entity to the comment. In the case that the ranking is even, the succeeding source code entity is chosen, since among developers, it is common practice to write comments preceding the associated source code entity or block.

In the example above, all the heuristics apply on both source code entities `foo()` and `bar()`. They are adjacent to the comment in between them and have the same textual similarity—both words, "foo" and "bar," are in the comment. Because the ranking is even, we choose the succeeding entity, i.e., `bar()`, as the associated entity. We show in Sect. 4.2.2 that these heuristics also perform well in practice.

3.2 Extracting comment changes

To extract comment changes with ChangeDistiller, we have to match comment nodes in the ASTs across subsequent revisions. The matching between comments is computed by a token-based similarity measure. This takes comment updates also into account, which an exact matching would not detect. To match two strings s_1 and s_2, the strings are first split into bags (multisets) of tokens, $T(s_1)$ and $T(s_2)$, according to a given non-alphanumeric separator. The similarity value of the two strings is then calculated as

$$sim(s_1, s_2) = \frac{|T(s_1) \cap T(s_2)|}{\max(|T(s_1)|, |T(s_2)|)}$$

We use this similarity measure because it is robust to rearrangement of the text in a comment.

To select an appropriate threshold to decide whether two comments are similar we extended our benchmark (Fluri et al. 2007) with comment changes. Overall, the most accurate results were obtained by defining that two comments c_1 and c_2 are similar if $sim(c_1, c_2) \geq 0.4$.

An update between two comments happens if they are similar enough to match but are not equal.

3.3 Relating comment changes to source code changes

Summarizing the steps described in the previous sections, we have gathered all data that we need to investigate whether or not comments are adapted when source code changes: (1) For each comment, we can compute to which source code entity it belongs, i.e., which source code entity it describes; (2) the change types describe when and how source entities as well as comments have changed. By combining (1) and (2), we can address:

1. Whether a comment and its associated source code entity changed at the same time or the comment changed later,
2. Whether the changes were of the same type (insert, delete, move, or update), and
3. Which source code change type is most likely to be part of a co-change

Consider the example chain of comment changes in Fig. 2. In Revision 1.2, a comment, `/*threshold at 0.8*/`, is inserted for the source code entity `double t = 0.8;` (variable declaration). The source code entity changes in Revision 1.3, but the corresponding comment is not updated until Revision 1.4. Both, comment and associated source code entity, are deleted in Revision 1.5.

We reconstruct such chains backwards by starting with the latest revision r_i. Attached on each revision is a set of source code and comment changes C_i. For each comment change $cc \in C_i$ we check if the associated source code entity was also changed. If the associated entity changed as well, we stop and store that there was a *co-change* between the comment and its associated entity, whether they changed the same way (i.e., insert, delete, move, or update), and the change type of the associated entity. In our example (Fig. 2), we start with the comment deletion in Revision 1.5. The associated entity and the comment changed in the same revision and in the same way.

If the associated entity did not change in r_i, we check for corresponding changes in r_{i-1}, thus going backwards. This step is repeated until we either find a change of the associated entity, or another change of the comment. In the former case, we store that there was a *shifted co-change* between the comment and its associated entity. If another change of the comment occurs, a new element in the chain begins, and we state that *cc* occurred without a source code change. In our example the comment in Revision 1.4 was changed one revision later than its associated entity. The comment insert in Revision 1.2 happened without a corresponding source code change.

The investigation of our example chain answers the third research question we posed at the beginning of this paper and its results can be summarized as follows: The comment

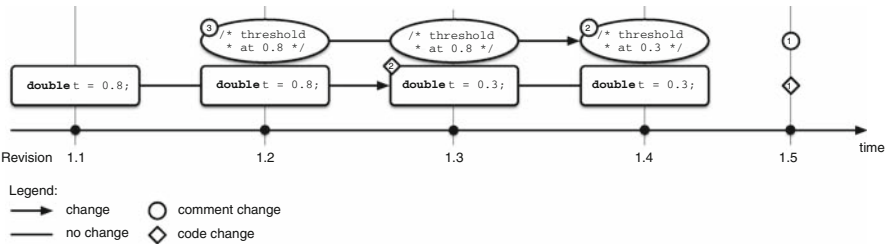


Fig. 2 An example of a chain of comment changes. Co-changes of source code and comments have the same number

changed three times. The last change (in Revision 1.5) happened in the same revision accompanied by a change of the associated entity of the comment. They had a co-change and both, the comment and the entity, changed the same way (delete). The second change (in Revision 1.4) occurred one revision later than the change of its associated entity, thus, they had a shifted co-change. The first comment change (in Revision 1.2) was applied solely. We can also state that it is more likely that a statement delete occurs together with a comment change in the same revision than that a statement update does.

We also check whether a comment describing a scope has changed due to source code changes inside the scope. For instance, when the body of a for-loop changed, it is likely that the comment describing the for-loop changed as well. Consider the following concrete example, where the insert of an if-statement triggered an adaption of the comment describing the for-loop:

```
- ->v.01
// calls execute() for all elements in the list
for (int i = 0; i < list.size(); i++) {
   list.get(i).execute();
}

- -> v.02
// calls execute() for all elements in the list,
// only if the element is ready (bug #13)
for (int i = 0; i < list.size(); i++) {
  if(list.get(i).isReady()) {
    list.get(i).execute();
  }
}
```

There are still open issues related to this concern: For scopes, we are unable to extract such shifted co-changes. Our change model stores source code entities only when they change. But to reconstruct co-changes over scopes, a complete source code model with unique identifiers is necessary. Since organizing and storing all this data suffers from a remarkable performance and storage overhead, we decided to skip it. For co-changes in the same version, we can leverage the information on source code location of the entities and comments to reconstruct the scoping.

To distinguish (a) comment changes that occur together with a change of their associated entity from (b) comment changes that happen together with changes inside the scope of their associated entity, we speak of *direct co-changes* and *scope co-changes*.

4 Empirical results

In this section we describe the results of our three independent empirical studies. In Sect. 4.1 we present the setup; in Sect. 4.2 we validate our data extraction and collection process; in Sect. 4.3 we describe the organization of the three empirical studies; in Sects. 4.4–4.6 we address the three research questions independently.

4.1 Setup

We conducted our empirical studies with eight software systems. These systems consist of three major components of Eclipse, one commercial system, and four open-source systems from different domains:

1. ArgoUML (UML designing tool). 14 releases: 0.9.6; 0.9.8; 0.13.2; 0.13.6; 0.14.0; 0.15.1; 0.15.6; 0.16.1; 0.17.1; 0.17.5; 0.18.0; 0.19.1; 0.19.8; 0.20a
2. Azureus (Java bittorrent client). 12 releases: 2.0.3; 2.0.4; 2.0.6; 2.0.7; 2.0.8; 2.1.0; 2.2.0; 2.3.0; 2.4.0; 2.5.0; 3.0.0; 3.0.1
3. Eclipse Core (21 plugins from the Eclipse platform component). 15 releases: 2.0.0; 2.1.0; 2.1.1; 2.1.2; 2.1.3; 3.0.0; 3.0.1; 3.0.2; 3.1.0; 3.1.1; 3.1.2; 3.2.0; 3.2.1; 3.2.2; 3.3.0
4. Eclipse JDT (17 plugins from the Java Development Tools component). 15 releases: 2.0.0; 2.1.0; 2.1.1; 2.1.2; 2.1.3; 3.0.0; 3.0.1; 3.0.2; 3.1.0; 3.1.1; 3.1.2; 3.2.0; 3.2.1; 3.2.2; 3.3.0
5. Eclipse PDE (5 plugins from the Plugin Development Environment component). 15 releases: 2.0.0; 2.1.0; 2.1.1; 2.1.2; 2.1.3; 3.0.0; 3.0.1; 3.0.2; 3.1.0; 3.1.1; 3.1.2; 3.2.0; 3.2.1; 3.2.2; 3.3.0
6. jEdit (text editor). 12 releases: 4.0.pre1; 4.1.pre1; 4.1.pre6; 4.2.pre3; 4.2.pre7; 4.2.pre11; 4.2.pre15; 4.2.final; 4.3.pre1; 4.3.pre2; 4.3.pre3; 4.3.pre5
7. JFreeChart (Java chart library). 10 releases: 0.9.21; 1.0.0.pre2; 1.0.0.rc1; 1.0.0.rc2; 1.0.0; 1.0.2; 1.0.3; 1.0.4; 1.0.5; 1.0.6
8. Webframework (a commercial framework for web applications). 13 yearly quarters.

All projects are written in Java and are version-controlled using CVS. ArgoUML, jEdit, and JFreeChart have already moved to Subversion. For jEdit we received an older repository dump directly from the developers, for ArgoUML we use the repository provided by the MSR Workshop Challenge of 2006, and for JFreeChart, the CVS repository is still available on sourceforge.net. Table 1 summarizes the software systems.

Table 1 Analyzed software systems

System (# developers)	# Source revisions	# Changes	# Comment changes (%)	LOC	
				First	Last
ArgoUML (41)	39,421	183,752	24,049 (13%)	200,735	239,791
Azureus (30)	33,008	245,214	13,790 (6%)	17,227	362,316
Eclipse Core (47)	15,454	69,383	9,714 (14%)	61,592	133,574
Eclipse JDT (55)	121,442	904,786	79,351 (9%)	420,233	974,006
Eclipse PDE (23)	35,137	153,891	6,534 (4%)	66,638	225,516
jEdit (18)	6,754	88,932	8,887 (10%)	80,726	133,895
JFreeChart (5)	4,675	23,678	3,166 (13%)	151,040	250,180
Webframework (33)	19,501	116,994	9,735 (8%)	43,452	124,796
Total	275,392	1,786,630	115,226 (9%)	1,041,643	2,444,074

The number in parentheses beside the system name indicates the number of developers taken from the versioning system. # Source revisions indicates the total number of revisions of Java files. # Changes indicates the number of changes type occurrences that were applied during the period. # Comment changes indicates the number of comment change type occurrences that were applied during the period. LOC first and last indicate the lines of code for the first and the last release of the component in the period

In the remainder of this section, we validate the parts of our data extraction and collection process (Sect. 4.2). Then, we explain the underlying rationale, formulate the hypotheses, perform a corresponding empirical study on the systems under investigation and discuss the results for each research question (Sects. 4.4–4.6).

4.2 Validation of data extraction and collection

For each step of the data collection process we validated its output to show the accuracy of the process. In this section we focus our validation on filtering commented source code, on the mapping between source code entity and comments, as well as on the tracking of co-changes between comments and source code. The change extraction validation is described in-depth in Fluri et al. (2007).

4.2.1 Comment filtering

We have randomly selected 8,978 comments from the latest releases of the eight software systems and inspected them manually to decide whether they are comments or commented source code. Out of these comments, we classified 372 as commented source code. Our simple pattern matching algorithm found 240 true positives, 87 false positives, and 132 false negatives leading to a precision of 0.73 and a recall of 0.65. Since simple regular expressions do not have the power of a parser but show a better runtime performance, these numbers of false positives and negatives are acceptable. Nevertheless, we expected to gain a higher recall. We found that 88 (66%) of the false negatives are due to comments at the beginning of files in JFreeChart. Our regular expression matcher filtered them as commented source code because such comments include code characters, such as '=' or '`]`', as delimiters and dotted expressions, such as `1.2.3`.

4.2.2 Comment to source code mapping

We have randomly selected 761 comment and source code mapping pairs. The manual inspection of these pairs revealed that (a) each comment was associated to a source code entity (0 false positives) and (b) 712 out of the 761 associations were semantically correct. Thus, our comment to source code association approach has a precision of 0.94.

By randomly selecting mapping pairs, we are not able to collect the false negatives because they are not in the set of mapping pairs. By counting the number of unmapped comments, we collected the false negatives for the whole data set. In total 258,555 comments were extracted from the eight software systems but 7,682 (3%) could not be mapped (false negatives). Over half of the false negatives (62%) are found in jEdit. Developers of jEdit block any type of scope (classes, methods, if-statements, etc.) with beginning and ending line comments: `//{{{Debugging` and `//}}}`. Our algorithm deals with triples of {source code entity, comment, source code entity} to decide whether a comment belongs to its preceding or succeeding entity. As such triples are missing at multi-level scope ends, the special jEdit comments are not mapped. Although the impact of this limitation is tremendous in jEdit, we decided not to overcome this situation in general because jEdit is an outlier compared to the other investigated systems. Removing jEdit from the data set leads to 253,778 comments in total, of which 2,905 (1%) could not be mapped.

4.2.3 Co-change tracking

We checked whether co-changes between comments and source code are tracked correctly, meaning whether the changes semantically correspond.

We have randomly selected 237 comment and source code co-change pairs. Out of these pairs, 221 were tracked correctly. Thus, our co-change tracking approach has a precision of 0.93. By randomly selecting co-change pairs, we are not able to collect false negatives because they are not in the set of co-change pairs.

4.3 Organization of empirical studies

We organize our studies and the discussion of the results according to the scheme of Baresi and Morasca (2007):

Question. This is the underlying research question that we want to answer.

Rationale. The reason why we claim that the research question is relevant.

Hypothesis. We outline the claim whose truth we want to check with our empirical analysis and describe the statistical hypothesis that we test.

Results. We present the results we obtain from the empirical studies and how we test the hypothesis.

Discussion. We discuss the results and reflect how the observations relate to the research question.

Summary of empirical study. We give a summary about the findings of the study.

4.4 Empirical study 1: growth factors of source code and comments

Question. Is the growth factor the same for source code and comments, meaning that about the same relative amount of code and comments is added over time?

Rationale. Software systems tend to become more mature with every release: The public API becomes more stable, most parts of the implementation have undergone several reviews, and re-documentation during maintenance takes places. This question is relevant when finding out whether developers tend to stop commenting their code once their system stabilizes. Answering the first research question therefore allows us to better understand the life cycle of software systems.

Hypothesis. We expect that the growth factor is the same for source code and comments. Let R_i be a release and R_j its succeeding release of a software system. $g_{comment_{ij}}$ is the growth factor of comments and $g_{source_{ij}}$ is the growth factor of source code between the releases R_i and R_j. If comments and source grow in the same proportion the difference $d_{ij} = g_{comment_{ij}} - g_{source_{ij}} = 0$. We formulate the following hypothesis to express our assumption of equality in growth between source and comment: The difference d_{ij} between any pair of subsequent releases R_i and R_j of a software system equals 0.

Results. We decided to use a *two-tailed one-sample t-test* to statistically verify whether our hypothesis holds or can be rejected. The *t-test* is adequate as significance test in our case because the variance σ is unknown and the size n of the sample set is <30. The test was performed under exactly the same setup for every software system shown in Table 1: We first extracted the set A of all $(g_{comment_{ij}}, g_{source_{ij}})$ pairs for any two subsequent releases R_i and R_j. We then calculated the set B of all differences d_{ij} for every $(g_{comment_{ij}}, g_{source_{ij}})$ pair in A and calculated $\bar{x}$ as the arithmetic mean for all d_{ij} in B where n is the size of the set B referring to the number of calculated differences d_{ij}. s is the standard deviation, i.e., the square root of the experimental sample variance calculated on B. We tested t against a

Table 2 Data of empirical study 1

System	$\bar{x}$	s^2	n	t	Accept ($\alpha = 0.01$)
ArgoUML	0.058	0.0104	13	2.04	Yes
Azureus	0.143	0.2789	11	0.90	Yes
Eclipse Core	0.012	0.0014	14	1.24	Yes
Eclipse JDT	−0.002	0.0007	14	−0.30	Yes
Eclipse PDE	0.012	0.0014	14	1.24	Yes
jEdit	0.014	0.0005	11	2.18	Yes
JFreeChart	−0.007	0.0007	9	−0.84	Yes
Webframework	−0.027	0.009	12	−0.98	Yes

$\bar{x}$ indicates the arithmetic mean for all d_{ij}. s^2 indicates the sample variance. n indicates the number of calculated differences d_{ij}. t indicates the calculated t-value

significance level $\alpha = 0.01$ resulting in a rejection region $t > t_{0.995,v}$, where $v = n - 1$ describes the *degrees of freedom* parameter of the *Student's t-distribution*. In Table 2 we list the numbers necessary for the t-test and show that our hypothesis is accepted in all cases.

Discussion. Figure 3 depicts the results of Study 1. For each system we plotted the number of non-commented lines (NCLOC) of code and number of comment lines (NCL) with a solid line. The dashed lines are the growth factor of NC-LOC (GF-NCLOC) and the growth factor of NCL (GF-NCL). The plots show the acceptance of our hypothesis. The course of the dashed lines mostly coincide. Although, both, source code and comments, grow equally in all investigated software systems, newly added code is barely commented in half of the systems: Azureus, Eclipse JDT, Eclipse PDE, and jEdit have only between 27% (Eclipse PDE) and 42% (jEdit) commented source code. In contrast, the systems ArgoUML, Eclipse Core, JFreeChart, and Webframework have between 53% (Webframework) and 100% (JFreeChart) commented lines of source code. Except for ArgoUML and Azureus, this commenting behavior is constant for all systems. For ArgoUML the commenting behavior was getting better, meaning that newly added code was commented more intensely after Release 0.15.6. Figure 3 depicts that for the Releases 0.15.6 and 0.16.1 of ArgoUML the growth factor of NCL is bigger than of NCLOC. The commenting behavior of Azureus is the opposite; after Release 2.1.0, source code was getting barely comment.

Since the core Eclipse IDE is mainly developed at IBM and because they have coding conventions[2] that are valid for all Eclipse components, the differences between the commenting behavior of the three Eclipse components is surprising. Eclipse Core is the only component that has a high commented source code ratio—90% on average. Eclipse JDT has a ratio of 40% and Eclipse PDE of 24% in average. A possible explanation for these discrepancies is that the ratio between public API and internal implementation in Eclipse Core is higher than in JDT and Core. Eclipse is known to have a comprehensive public API documentation, but a modest internal implementation documentation, as confirmed by Schreck et al. (2007). The second study shows which type of source code is more likely to be commented. This will explain the differences in the commenting behavior of the three Eclipse components.

[2] http://wiki.eclipse.org/Coding_Conventions.

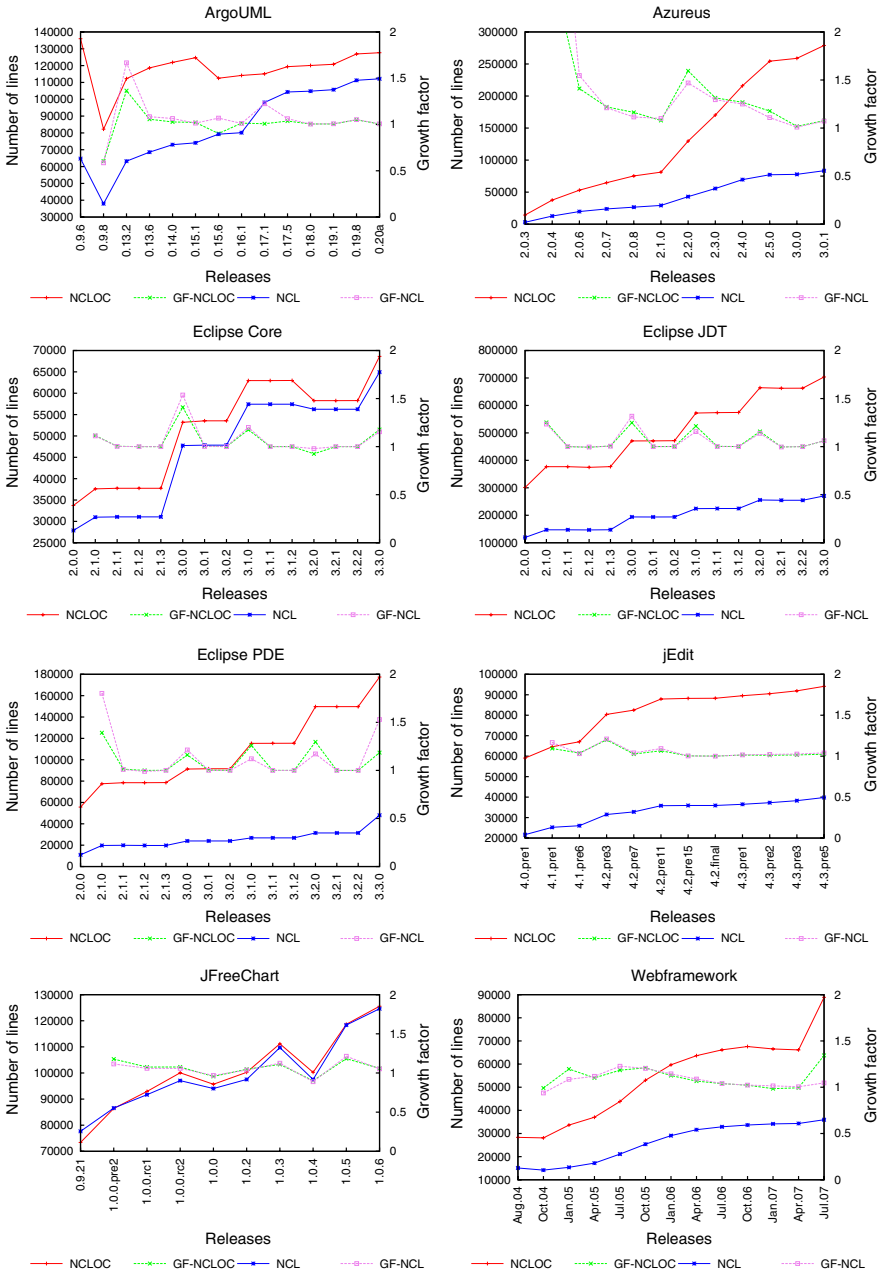


Fig. 3 Result of empirical study 1. NCLOC and NCL indicate the number of non-commented lines of code and number of comment lines. GF-NCLOC and GF-NCL indicate the growth factor of NCLOC and NCL between two subsequent releases. The growth factor curves are *dashed*

The reason for the high percentage of commented source code in JFreeChart are the long file header comments, but also the intensive API documentation—an exemplar of well documented software.

Summary of empirical study 1. We have statistically shown that source code and comments grow equivalently over time in all eight software systems. We have also shown that this does not directly mean that newly added code is well commented in all systems. Half of the investigated systems have a commented source code ratio of less than 50%. Even systems that are developed in the same company, such as IBM for Eclipse, have different commenting characteristics. To conclude, equal growth factors are an indication that the ratio of commented source code remains stable. But, it does not mean that newly added code is commented comprehensively.

4.5 Empirical study 2: commented source code entities

Question. Does the type of the source code entity have an influence on whether it gets commented and which source code entities are most likely to be commented?

Rationale. Do all source code entities have the same likelihood of being commented? Or more precisely, is there any statistical evidence that programmers are more likely to add documentation to building blocks of a program, such as class or method declarations, or to if and loop scopes, rather than to simple statements? These questions are relevant because the answer indicates whether developers are aware that commenting declaration parts and scope increases readability and makes programs more comprehensible and therefore easier to maintain in the long-term.

Hypothesis. We claim that the type of a source code entity has an influence on whether it gets commented or not. The statistical hypothesis we test is, whether source code is commented or not is independent from the source code entity type.

Results. We decided to use the *two-variable* χ^2*-test* to statistically verify our hypothesis. The χ^2*-test* evaluates whether observed frequencies reflect the independence of two qualitative variables. In our data set, the first variable describes whether a source code entity type is commented or not. The second variable describes the source code entity type.

For each software system shown in Table 1 we calculated the (observed) numbers of commented as well as non-commented source code entity types and the corresponding expected values of the latest release. The expected values are those values we would expect if we assume that the number of commented source code entities does not depend on their type, i.e., commented source code entities are proportionally equally distributed among the different types. Table 3 shows the observed frequencies as well as the expected values of commented source entities for ArgoUML grouped by their type. The column *other* refers to all source code entities that do not fall under the explicitly listed categories, such as

Table 3 Observed (Obs.) and expected (Exp.) contingency tables of Release 0.20a of ArgoUML

Obs.	Class	Field	Method	If	Loop	Var.-Decl.	Call	Other	Total
c	1,659	1,606	12,347	765	81	1,034	744	595	18,831
$\bar{c}$	33	1,852	0	9,469	1,531	10,577	17,255	42,832	83,414
Total	1,692	3,458	12,212	10,234	1,612	11,611	17,999	43,427	102,245

Exp.	Class	Field	Method	If	Loop	Var.-Decl.	Call	Other
c	312	637	2,249	1,885	297	2,138	3,315	7,998
$\bar{c}$	1,380	2,821	9,963	8,349	1,315	9,473	14,684	35,429

c indicates the number of commented source code entity types. $\bar{c}$ indicates the number of non-commented source code entity types. The numbers in the expected contingency tables are rounded

try-catch-statements, infix expressions, assignments, etc. Using the χ^2-*statistic* we can test if the observed values differ significantly from the expected values under the assumption of independence.

We tested χ^2 against a significance level $\alpha = 0.005$ resulting in a rejection region $\chi^2 > \chi^2_{0.995,v=7}$, where $v = (r-1)\cdot(c-1)$ is the degree of freedom, a function of the number of rows, r, and the number of columns, c, of the contingency table.

The χ^2 values of the eight software systems are all >11,600. Since $\chi^2_{0.995,7} = 20.278$ the hypothesis is rejected and we conclude that whether a source code entity gets commented or not depends on its type.

Discussion. The results of the statistical tests show that the source code entity types do not have the same likeliness for being commented in all investigated software systems. We expected this result because commenting high level scopes, such as methods or classes, has a higher impact for understanding software than lower level scopes or simple statements.

Figure 4 depicts the results of Study 2. According to the diagrams in this figure, there is a partial order in the likeliness whether a certain type of source code entity gets commented or not. *The high level scope, class, method, and field, are more commented than the low level scope or simple statements*.

Except for jEdit and JFreeChart, the order of high level scopes changed over time. In jEdit methods are more commented than classes and fields. In JFreeChart each class, method, and field was commented in all releases. JFreeChart is an exemplar for well documented API. In Azureus and Eclipse PDE commenting the API was neglected from the beginning of the project. The percentage of commented classes decreased in both systems towards the latest release; in Azureus the same characteristics applies to the methods as well. All other systems either have a stable or increasing percentage.

Low level scopes and simple statements are barely commented in all systems; the highest percentages are found in Eclipse Core (<16%). In all other systems they are below 10%. The order of commenting low level scopes and simple statements varies for the eight systems. If-statements are commented most frequently for five of them; in ArgoUML and jEdit the most often commented low level entity type are variable declaration statements, in JFreeChart the loop statements. Except for ArgoUML, calls are the least commented low level entity types.

The three diagrams of the Eclipse components show the reason why the ratio of commented source code is higher in Eclipse Core than in Eclipse JDT and PDE. In Eclipse Core classes and methods are about 20% more often commented than in Eclipse JDT and about 45% more often than in Eclipse PDE. Low level scopes and simple statements are also more often commented in Eclipse Core than in JDT or PDE, but not to that extent as high level scopes.

Overall, the only system that consistently comments high level scopes is JFreeChart. ArgoUML and partly jEdit have at least high commenting percentages for methods and classes. For all other systems, these percentages hardly go over 80%. One of the reasons for the low high level scopes commenting is that often private members are not commented.

Summary of empirical study 2. We have statistically shown that whether a source code entity gets commented or not depends on the type of entity. We have also shown that there is a partial order in the likeliness of whether a certain source code entity gets commented. High level scopes, such as classes, methods, or fields are more likely to be commented than low level scopes and simple statements, such as if-statements or calls. During the history of five of the investigated software systems the commenting percentages stayed stable, in two they increased and in one they partly decreased. A consistent commenting behavior is only identifiable in JFreeChart. Although it is well-known that comments describing low level scopes increase understandability, they are barely commented in all investigated systems.

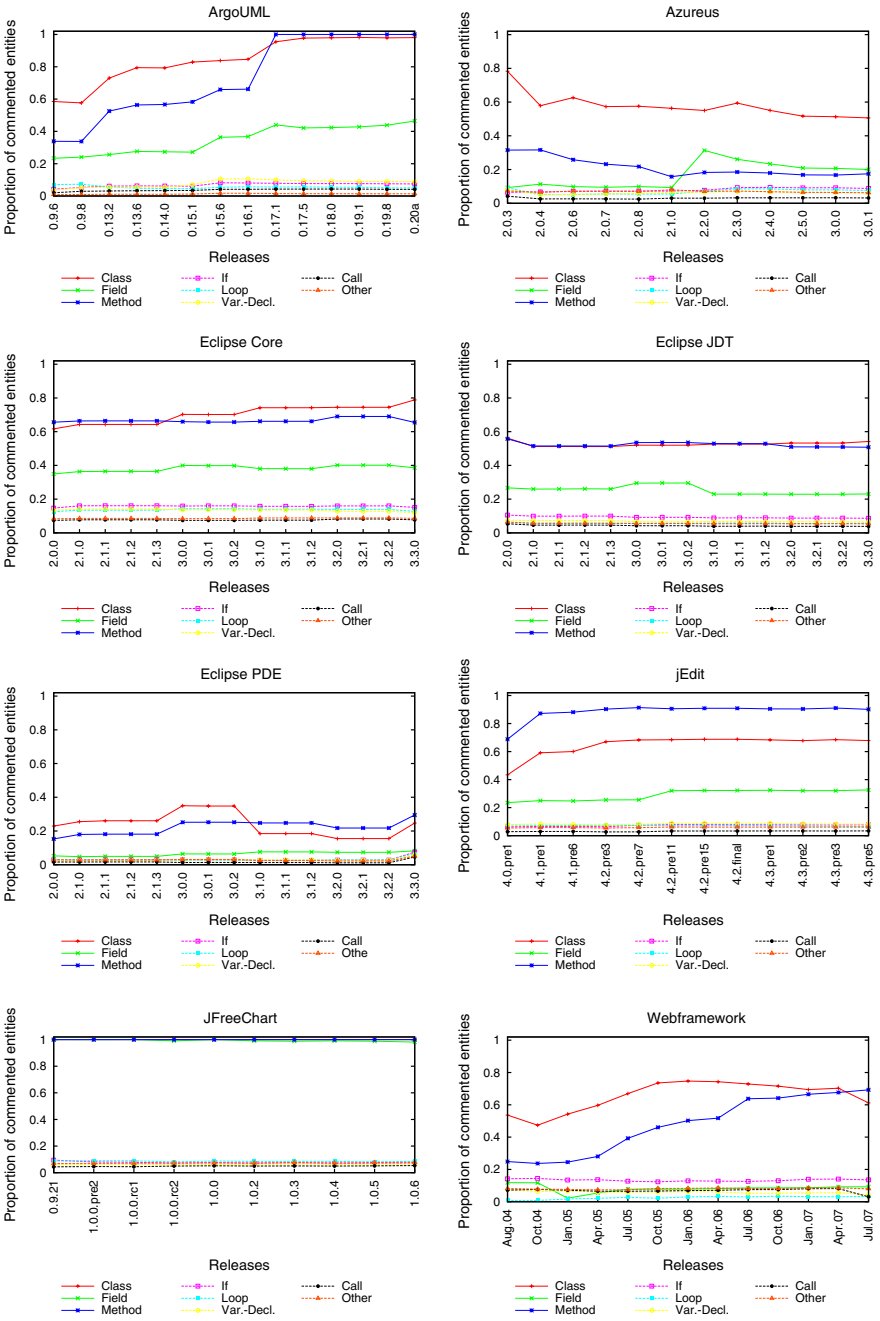


Fig. 4 Result of empirical study 2. For each release and source code entity type of the investigated software systems we plot the proportion of commented source code entity types. The curves of the high-level constructs (class, method, and field) are *solid*, those on statement levels (if-statement, loop-statement, variable declaration, call, and other) are *dashed*

4.6 Empirical study 3: co-change of comments and source code

Question. Are comments adapted when source code is changed (i.e., are comments kept up-to-date) and when does the adaptation take place—while changing the source code or in a later revision?

Rationale. By answering this question, we can draw conclusions about whether re-documentation is an integral part in the software engineering process, even though programmers often neglect to adapt documentation to source code changes immediately. In other words, we want to analyze if development in general follows a cycle similar to *apply bug or feature request → commit source code changes → adapt documentation → commit comment changes → ····*?

Hypothesis. To keep comments up-to-date, we expect that the majority (i.e., >50%) of the comment changes were related to changes of their associated source code entity in the same revision.[3]

Results. To answer the third question, we have calculated chains of comment changes for each system. The results can be found in Table 4. The column "co-change" includes both, co-changes and shifted co-changes. To recapitulate, we speak of a *direct* (shifted) co-change if the comment change is related to a change of its associated source code entity; and we speak of *scope* co-change if the comment change is related to a source code change inside the scope of its associated entity. Scope co-changes include scoped statements as well as declarations whereas shifted scope co-changes only include declarations.

The hypothesis is accepted for a software system, if the "co-change" column multiplied with the "$\Delta_r = 0$, both" column is >50%. This multiplication indicates the percentages of comment changes that happened together with changes of their associated entity in the same revision. As the results in Table 4 show, we can accept the hypothesis for four software systems: Azureus, Eclipse JDT, Eclipse PDE, and JFreeChart. For the other four systems less than 50% of comment changes were related to changes of their associated entity in the same revision. We discuss reasons for these results in Sect. 5.2.

Discussion. There is a significant difference in the behavior of direct and scope co-changes. During the evolution of all systems, 98% of direct co-changes happen in the same revision; there are only a few shifted direct co-changes. In contrast to the direct co-changes, between 57% (Eclipse Core) and 93% (jEdit) of scope co-changes happen in the same revision. We can also observe that shifted co-changes between API comment and declarations occur: In Eclipse Core, for instance, in 10% of the scope co-changes the comment changed one revision after the scope changed. And, in 23% of the scope co-changes the comment changed more than three revisions after the scope changed. Systems that have shifted scope co-changes either have a significant amount of shifted co-changes in $\Delta_r = 1$ or $\Delta_r > 3$. That means, the comment is either adapted shortly after the code is changed or long after the code has changed; for instance during a consolidation phase before a release.

For all direct co-changes and direct shifted co-changes we have calculated the proportions of source code change types that are related to comment changes. We distinguish between change types in the method body (body change types) as well as class, attribute, and method declaration change types (declaration change types). These proportions are listed in Table 5.

[3] Compared to the hypotheses of Study 1 and 2 this is rather an assumption than a statistical hypothesis. Nevertheless we use the term hypothesis to keep the organization of the empirical studies consistent.

Table 4 Data of empirical study 3

System	Co-change	$\Delta_r = 0$			$\Delta_r = 1$		
		Both	Direct	Scope	Both	Direct	Scope
ArgoUML	50.66	81.58	98.14	62.55	4.69	0.22	9.87
Azureus	68.80	97.91	98.77	90.77	0.51	0.29	3.05
Eclipse Core	53.26	89.23	99.02	57.23	2.53	0.10	10.49
Eclipse JDT	65.93	93.70	98.83	84.95	1.31	0.18	3.39
Eclipse PDE	61.03	93.44	98.66	66.15	1.51	0.89	7.87
jEdit	50.06	96.67	98.85	92.98	0.54	0.29	1.02
JFreeChart	56.47	91.49	99.49	75.26	4.20	0.17	12.37
Webframework	52.73	91.31	98.76	72.93	1.83	0.16	6.15

System	$\Delta_r = 2$			$\Delta_r = 3$			$\Delta_r > 3$		
	Both	Direct	Scope	Both	Direct	Scope	Both	Direct	Scope
ArgoUML	2.94	0.16	6.13	1.83	0.23	3.66	8.97	1.26	17.79
Azureus	0.35	0.16	1.74	0.20	0.07	1.13	1.03	0.72	3.31
Eclipse Core	1.28	0.10	5.12	1.16	0.20	4.29	5.80	0.58	22.87
Eclipse JDT	0.74	0.06	1.86	0.58	0.05	1.47	3.66	0.87	8.32
Eclipse PDE	1.11	0.12	5.62	1.31	0.06	7.02	2.62	0.27	13.34
jEdit	0.36	0.25	0.54	0.34	0.14	0.66	2.10	0.47	4.80
JFreeChart	1.76	0.17	4.98	1.08	0.00	3.26	1.47	0.17	4.12
Webframework	1.89	0.05	6.35	0.94	0.19	2.74	4.03	0.82	11.83

Co-change indicates the proportion of comment changes that happen together with source code changes in the same revision or later. Δ_r indicates the number of revisions that elapsed between the source code and the comment change. We distinguish between direct and scope co-changes. Column "both" merges "direct" and "scope." The values in the table are in %. The percentages of these columns are relative to the "co-change" column. For instance, for ArgoUML 81.58% of all co-changes happen in the same revision

The proportion of body to declaration change types that happened together with comment changes are related to the results of Study 2. The more classes, fields, and methods are commented the higher is the proportion of declaration change types that are related to comment changes. JFreeChart and Webframework have the highest proportion of declaration change types inducing comment changes, whereas comment changes in Azureus or Eclipse PDE are mostly related to body change types.

Figure 5 shows the distribution of body and declaration change types that are related to comment changes. In all software systems the change types *statement insert* and *statement delete* were encountered most often together with comment changes. It is not surprising that statement delete has such an influence on comment changes because when a developer deletes a statement, she can delete the corresponding comment right away. Statement insert is one of the most applied change types and, therefore, it is obvious that it is also observed mostly together with comment changes. The change types *statement parent change* and *statement update* occur also with comment changes. It is surprising that statement updates have a marginal influence on comment changes. We expected a stronger relation because statement updates are applied often, and after such a change a comment might be outdated. A similar argument is valid for the surprising observation that *condition expression*

Table 5 The proportions of body and declaration change types that occurred in direct and direct shifted co-changes

Project	Body (%)	Declaration (%)
ArgoUML	72	28
Azureus	92	8
Eclipse Core	76	24
Eclipse JDT	81	19
Eclipse PDE	87	13
jEdit	77	23
JFreeChart	32	68
Webframework	66	34

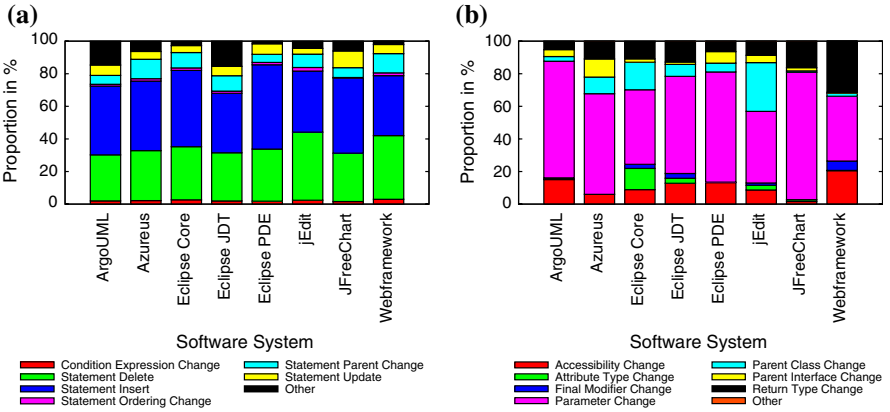


Fig. 5 Distribution of body (**a**) and declaration change types (**b**) that are related to comment changes

changes do not have a significant influence on comment changes. The change type *other* means that we still encountered co-changes between comments. As we explain in Sect. 5, we still have issues with successive comments that are in different scopes.

Parameter changes are responsible for the most API comment changes in all investigated software systems. This is obvious because parameters are mostly described in API comments with a corresponding tag. In addition, IDEs, such as Eclipse, support the adaptation of such tags when declarations are changed. *Return type changes* influence changes in the API comment because they also have a predefined tag. A reason that *accessibility changes* have a significant influence on API comment changes may be as follows. Assume the modifier `private` of a method is changed into `public`. Because of that, either a API comment has to be added to describe the method, or the API comment has to be complemented with more detailed information. Changing the parent class or parent interface of a class has partly an influence on changing API comments.

Summary of empirical study 3. Over 50% of the comment changes are related to source code changes. We have shown that direct co-changes happen in over 98% of the cases in the same revision. But we observed that API comments are more often adapted retroactively than other comments. This seems reasonable under the assumption that there are often public interfaces involved, which are more likely to be subject of re-documentation.

Source code change types that are related to comment changes can be split into body and declaration change types. Statement insert and delete are the body change types that

are most often accompanied by comment changes. Parameter changes, return type changes, and parent class or parent interface changes are the declaration change types that induce the most API comment changes.

5 Discussion

Our investigations of comments and their changes expose interesting insights of the commenting behavior and process of software systems. In this section, we report on how we can leverage these investigations in terms of software quality. We also discuss whether our comment to source code mapping approach is appropriate.

5.1 Interpretation in terms of software quality

Comments describe the source code of a software system. If they exist and are meaningful, they can aid in comprehending the system. In addition, meaningful comments allow us to reason about the source code and aid assessing its quality. Based on our investigations of comments and their change behavior we contribute to the quality assessment of comments. With our approach we can assess the commenting behavior quantitatively and reflect on the commenting process of software systems. The results of our analysis can be compared against those for other projects and serve as an assessment for a particular aspect of the quality of a software and its development process.

For example, two of our industrial partner asked us to perform a quality analysis of their software systems. One of them was the company developing the Webframework.[4] Among other investigations, we suggested analyzing the commenting process of their projects and successfully applied the corresponding quality assessment. We briefly report on these experiences as well.

5.1.1 Assessing the comments quantitatively

Studies 1 and 2 assess the quality of the commenting behavior on a quantitative basis.

Empirical study 1. Comparing the growth factor of the number of comment lines and the number of non-commented lines of code shows whether the proportion of comment lines to code lines increased, decreased, or stayed stable over the history of a software system. This neither indicates that the source code is well commented nor that the comments are meaningful. But it shows whether or not developers of a system comment their code consistently over time.

Empirical study 2. The results of Study 1 give an impression on the amount of comments in a software system. Conducting this study is straightforward and can be done with modern IDEs on the fly. But, simply counting the lines of code and the comment lines hides two major aspects of commenting: First, dead code is counted as comment lines. Second, which source code entity types are commented is not considered. We complement the interpretation of the results of Study 1 with Study 2 because dead code harms the comprehension of source code, and it makes a difference which and to what extent a certain source code entity type is commented to measure the quantitative quality of the comments. The less dead code is present and the more declaration parts as well as scopes are commented, the better the quantitative quality of the comments and the higher the maturity of the system.

[4] The detailed results of the other study are not available for publication.

Experiences with industrial partners. We have followed the development of the Webframework since April 2005 and periodically assessed the quality of the source code. At the beginning of this evaluation, we regarded the overall percentage of commented source code as sufficient but suggested improving the quantity of API comments. The company agreed on this quality factor and increased the proportion of commented methods and classes as Fig. 4 shows—our toolset allowed us to quickly assess the improvements quantitatively.

We also applied our investigations on the second commercial software system. The results of Study 1 gave the impression of a sufficient commented system. But the Study 2 showed that a lot of dead code was present at that time. Moreover, comments for declaration parts and scopes hardly existed. The answer of the company on our report was that they are payed for a *working* software system and *not for documentation* and that commercial projects in general cannot spend much effort on source code documentation. A comparison with data from other systems, however, convinced the development team that their quantity of comments nevertheless lags behind industrial standards. Again, our analysis proved itself useful to identify weaknesses in terms of software quality and provide reference data to assess the deficiencies in contrast to other projects.

5.1.2 Assessing the commenting process

To understand source code and prevent bugs, it is important to keep comments up-to-date (Tan et al. 2007). With our third empirical study we assessed whether comments are kept up-to-date or at least adapted several revisions after the associated source code entity changed. That shows whether re-documentation is an integral part of the development process. For instance, we found that in ArgoUML, Eclipse Core, Eclipse PDE, and the Webframework re-documentation for declaration parts took place (see Table 4). The sooner the comments are adapted to source code changes the better we assess the commenting process of a system. But we also approve re-documentation because source code comments are added better late than never.

It is not true that every change induces a comment adaptation. In particular, different change types impact the consistency between comments and source code differently. Source code change types let us assess whether developers are aware of these different impacts. As the impact factor we use the *change significance level* that is assigned to each change type (Fluri and Gall 2006). Concerning changes of scope comments we sum up the change significance levels of the change types applied inside the scope. The higher the significance level the higher the probability that the comment has to be adapted and the sooner this should take place.

Experiences with industrial partners. In the Webframework we found that a significant amount of declaration parts were re-documented. We know that the company employed a person mainly for the re-documentation. This decision enhanced the quality of the commenting process.

The company of the other mentioned software system does not re-document declaration parts. The statement of the company was that as soon as the code works, it is not touched anymore—whether the API is commented or not does not matter.

5.1.3 Feedback during evolution

Beside the assessment of the quality of the commenting behavior of a software system, we further benefit from our investigations. We can provide feedback during evolution with a

recommendation that suggests when a developer might adapt the comments to source code changes. The change significance levels are then used to decide whether a suggestion is appropriate and to give a certain level of confidence. For instance, assume a developer makes several changes in an if-statement. When the sum of the change significance levels exceeds a specified threshold we can automatically suggest to adapt the comment of the if-statement (unless one exists).

5.2 Threats to validity

According to Yin (2003) we structure our threats to validity section into *construct validity, external validity*, and *reliability*. Threats to *internal validity* did not affect our empirical studies because they are mainly explorative.

5.2.1 Threats to construct validity

To map comments to source code, we have chosen a set of heuristics as described in Sect. 3.1. The heuristics are straightforward, easy to understand, and reflect common practice, as confirmed by industrial partners. Nevertheless, we discuss issues of the mapping that might have influenced the results of our investigations.

Mapping comments to single source code entities. Our approach maps a comment to single source code entities. The limitation of this methodology is that not every comment describes a single source code entity. Developers also use comments to describe source code blocks, e.g., sequences of statements. We partly cover this practice by treating changes to scope comments but we miss sequences of simple statements. Consider the following illustrative example

```
// Button to save using 'this' as selection listener
Button button = new Button(parent, SWT.NONE);
button.addSelectionListener(this);
button.setText("Save");
```

The comment describes a sequence of three statements. Our approach maps the comment to the variable declaration statement. A source code co-change for the comment only happens when the variable declaration statement changes together with the comment. There is no co-change, when, for instance, the selection listener is changed:

```
// Button to save using OpenFileChooser as selection listener
Button button = new Button(parent, SWT.NONE);
button.addSelectionListener(new OpenFileChooser());
button.setText("Save");
```

There are two possible solutions to overcome this situation. First, additional line delimiters that format the source code can split sequences of statements. Second, we may use the assumption that statements in-between two comments are described by the first comment. However, both possibilities are inappropriate to implement. Using delimiters or comments to split related statements depends on the coding conventions of a development team in general and on the practices of a single developer in particular. Extracting these conventions manually—even automatically—is not feasible. Moreover, there is no

guarantee that such conventions or practices are applied consistently. This additional uncertainty factor would harm the validity of the results tremendously.

Incomplete mapping of comments to source code. Due to our approach of processing triples, we are currently not able to establish a proper mapping whenever successive comments are in different scopes. We always expect that a comment is among two source code entities; either on the class body or on the method body level. If not, then comments are related to comments, and comment changes to comment changes instead of source code changes.

This drawback results in comment changes that are due to comment changes. The change type *other* in Fig. 5a shows the percentages of such comment changes. In detail, these proportions are 15% for ArgoUML, 6% for Azureus, 3% for Eclipse Core, 15% for Eclipse JDT, 2% for Eclipse PDE, 4% for jEdit, 6% for JFreeChart, and 2% for Webframework. For each comment change the change of its associated source code entity is counted; changes between comments are therefore counted twice. Hence, the resulting error rate is between 1% and 8%, which we consider acceptable for such an empirical study. On the declaration level, the change type *other* does not have any impact (see Fig. 5b).

Not all comment changes are part of a co-change together with source code changes. For all systems, over 50% of all comment changes were found together with changes of their associated source code entities. We cannot expect that all comment changes are related to source code changes because of an external and an internal factor.

- *External factor*: Assume an interface without any API comment is added into the CVS repository and receives Revision 1.1. Its source code was not changed in Revision 1.2 but each method declaration was commented with an API comment. These inserts are not related to any source code change because no changes are recorded for Revision 1.1. A similar scenario can happen for other source code entities and normal comments as well.
- *Internal factor*: Reconsider the `Button` example that explained the association-issue. In that example the comment change was not co-changed with a source code.

Analyzing whether the internal or the external factor has a higher impact on the number of comment and source code co-changes is subject of future work.

5.2.2 Threats to external validity

There are two major external threats to the validity of this work that might affect the generalizability of our results.

Systems examined might not be representative. We examined eight software systems from different domains including one commercial system. It is still possible that we have chosen an unrepresentative set of systems for our study. However, the chosen open-source systems are well-known systems in the software evolution research community—especially ArgoUML, Azureus, Eclipse, and jEdit. Moreover, they have a rather long version history (3–7 years) to show a certain consistency in the commenting behavior. We found similarities as well as a certain diversity in the results of the three studies. It is even appropriate to use three different components from the Eclipse software system. Eclipse is developed all over the world and is big enough to have a diversity in development. Consider the number of developers of the three components: Core has 47, JDT 55, and PDE 23 authors. The overlap of developers between them is as follows: Core and JDT have 19 common developers, Core and PDE have 10 common developers, and JDT and PDE have 7 common developers.

Our result indicate that the commenting process of the commercial system is comparable to the commenting process of the open-source systems.

All systems are written in Java. Extracting source code and comment changes on the AST level requires a complete programming language parser. As a result ChangeDistiller currently supports the Java language. Systems in other object-oriented programming languages may be commented differently. However, we claim that the investigation of the commenting behavior of software systems is independent from the object-oriented programming language because common object-oriented languages provide similar language constructs for adding comments and commenting source code either depends on the development conventions or on the mood of developers.

5.2.3 Threats to reliability

Our studies can be repeated because of the following reasons: (1) except for the Web-framework (proprietary) the histories of the investigated software systems are publicly available. (2) the change distilling algorithm is described thoroughly in Fluri et al. (2007) and its implementation is available upon request. (3) We describe the conditions and settings necessary to conduct our statistical test in detail (see Sect. 4).

6 Related work

Work related to our change distilling algorithm (i.e., other source code change extraction algorithms) has already been discussed extensively in Fluri et al. (2007). Other areas of research relevant are investigations on commenting and documentation of software.

In his early experiment Tenny (1988) showed that comments had an influence on the readability of a *PL/1* program. The (positive) effect was most significant when the program did not contain any procedures. Dromey (1995) proposes a model to defining software product quality based on a set of *quality-carrying* properties. Comments are considered as a property that has an influence on maintainability, portability, reusability, and usability. In Hyatt and Rosenberg (1996) a *Software Quality Model* is proposed as a basis for the discussion on quality and risks of a software system. They refer to documentation as an important objective in software development as it is critical for understanding the software. A measure *Documentation* is established, i.e., the adequacy of internal code documentation and external documentation. Internal documentation is measured by *Comment Percentage*: #comments/(total lines of code − blank lines). Lucia et al. (2006) used traceability links between source code and documentation to guide the user in choosing meaningful identifiers and comments. In a controlled experiment they showed that their approach helps to improve the similarity between code and related requirements in presence of comments.

Our work focuses on the evolutionary characteristics of comments in the context of software quality.

Jiang and Hassan (2006) conducted a study on the evolution of comments in PostgreSQL. They investigated how many header comments and non-header comments were added or removed to PostgreSQL over time. In contrast to their work, we do not restrict ourselves to studying the addition and deletion of comments, but also track updates and moves. Moreover, we integrate source code change analysis down to the statement level to track whether and how source code and comments change together.

Antoniol et al. (2002) proposed an approach based on information retrieval to recover traceability links between source code and free text documents. Marcus and Maletic (2003)

proposed a similar solution. However, both approaches focus on external documentation and do not investigate evolutionary aspects, i.e., they do not track documentation and source code changes together over time. The issue of the evolution of traceability links between source code and documentation is discussed in Lucia et al. (2007). Recently Witte et al. (2007) used Semantic Web Technologies to connect software and documentation artefacts. They developed ontologies to query the linking. Information retrieval techniques were also employed by Lawrie et al. (2006) to measure how the comments relate to the source code and assume that comments impact the code quality of software systems. Marcus and Poshyvanyk (2005) defined metrics for measuring the conceptual cohesion of classes. For that, they incorporated the presence (absence) of comments.

Ying et al. (2005) investigated the usage of a particular type of comment, the Eclipse task comments, i.e., special comments starting with `// TODO` which are common used by developers using the Eclipse IDE. They argued that task comments tend to depend a lot on the context of the surrounding code and that it is difficult to infer the scope of a task comment. This often holds for comments in general and has therefore an impact on our work. Ying et al mentioned a few reasons that lead to an insert of a comment task (for example as pointers to change requests) but they did not study whether some building blocks of a program (e.g., if-statement) are more likely to be commented. Again, they did not analyze any evolutionary aspects, neither of source code nor comment. Schreck et al. (2007) analyzed the quality evolution of comments in the Eclipse project. They used metrics, such as completeness and quantity, to measure properties of comments. With their approach, they found, for instance, strong jumps in the documentation quality of Eclipse—an indication for re-documentation.

Whether false comments may have any impact on bugs was analyzed by Tan et al. (2007). They extracted implicit program rules out of comments to automatically detect inconsistencies between comments and source code. For that, they used natural language processing and machine learning. With this approach, Tan et al. found new bugs in several open-source C projects.

7 Conclusions

In this paper, we investigated how developers maintain source code documentation in the form of comments. In particular, we examined the question whether developers comment their code and to which extent they add comments or adapt them when they evolve the code. Our approach associates comments with source code entities to enable a tracing of their co-evolution over multiple versions. A set of heuristics are used to decide whether a comment is associated to its preceding or its succeeding source code entity.

We analyzed the co-evolution of code and comments in eight different open source and closed source software systems. We found with statistical significance:

1. Source code and comments grow equivalently over time in all eight software systems. This does not mean that newly added code is well commented; but half of the investigated systems have a comment to source code proportion of less than 50%.
2. It depends on the code entity whether it gets commented or not. We even observed a partial order in the likeliness of whether a certain source code entity get commented.
3. In six out of eight systems, the comment and its associated source code co-changed in more than 90% of all comment changes. Surprisingly, API changes and comments do not co-evolve but are re-documented in a later revision.

The results have shown that our approach enables a quantitative assessment of the commenting habits and the commenting process in a software system. We have successfully applied our tool in industrial projects to draw conclusions on different documentation-related aspects. As a result, we can leverage the results to provide feedback during software development to increase the awareness of when to add comments or when to adapt comments because of source code changes.

For future work we plan to associate comments with building blocks of code and consider the size of comments among different source code entities. In addition, we intend to conduct a time series study to observe co-change trends over the history of a software system.

Acknowledgements This work was supported by the Hasler Foundation as part of the ProMedServices—Proactive Software Service Improvements project. The authors would like to thank the reviewers for their insightful suggestions that greatly helped to improve the paper.

References

Antoniol, G., Canfora, G., Casazza, G., Lucia, A. D., & Merlo, E. M. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering, 28*(10), 970–983.

Baresi, L., & Morasca, S. (2007). Three empirical studies on estimating the design effort of Web applications. *ACM Transactions on Software Engineering and Methodology, 16*(4), 15.

Bevan, J., James Whitehead, E. J., Kim, S., & Godfrey, M. W. (2005). Facilitating software evolution research with Kenyon. In *Proceedings of the joint 10th European software engineering conference and the 13th ACM SIGSOFT symposium on the foundations of software engineering* (pp. 177–186). ACM.

Demeyer, S., Ducasse, S., & Nierstraz, O. (2003). *Object-oriented reengineering patterns*. Morgan Kaufmann.

des Rivières, J., & Wiegand, J. (2004). Eclipse: A platform for integrating development tools. *IBM Systems Journal, 43*(2), 371–383.

Dromey, R. G. (1995). A model for software product quality. *IEEE Transactions on Software Engineering, 21*(2), 146–162.

Elshoff, J. L., & Marcotty, M. (1982). Improving computer program readability to aid modification. *Communications of the ACM, 25*(8), 512–521.

Fischer, M., Pinzger, M., & Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *Proceedings of the 19th international conference on software maintenance* (pp. 23–32). IEEE Computer Society.

Fluri, B., & Gall, H. C. (2006). Classifying change types for qualifying change couplings. In *Proceedings of the 14th international conference on program comprehension* (pp. 35–45). IEEE Computer Society.

Fluri, B., Würsch, M., Pinzger, M., & Gall, H. C. (2007). Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering, 33*(11), 725–743.

Goldberg, A. (1987). Programmer as reader. *IEEE Software, 4*(5), 62–70.

Hyatt, L. E., & Rosenberg, L. H. (1996). A software quality model and metrics for identifying project risks and assessing software quality. In *European space agency software assurance symposium and the 8th annual software technology conference* (p. 209).

Jiang, Z. M., & Hassan, A. E. (2006). Examining the evolution of code comments in PostgreSQL. In *Proceedings of the 3rd international workshop on mining software repositories* (pp. 179–180). ACM.

Kaelbling, M. J. (1988). Programming languages should NOT have comment statements. *ACM SIGPlan Notices, 23*(10), 59–60.

Lakhotia, A. (1993). Understanding someone else's code: Analysis and experience. *Journal of Systems and Software, 23*(3), 269–275.

Lawrie, D. J., Feild, H., & Binkley, D. (2006). Leveraged quality assessment using information retrieval techniques. In *Proceedings of the international conference on program comprehension* (pp. 149–158). IEEE Computer Society.

Lucia, A. D., Penta, M. D., Oliveto, R., & Zurolo, F. (2006). Improving comprehensibility of source code via traceability: A controlled experiment. In *Proceedings of the 14th international conference on program comprehension* (pp. 317–326). IEEE Computer Society.

Lucia, A. D., Fasano, F., Oliveto, R., & Tortora, G. (2007). Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology, 16*(4), 50.

Marcus, A., & Maletic, J. I. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th international conference on software engineering* (pp. 125–135). IEEE Computer Society.

Marcus, A., & Poshyvanyk, D. (2005). The conceptual cohesion of classes. In *Proceedings of the 21st international conference on software maintenance* (pp. 133–142). IEEE Computer Society.

Schreck, D., Dallmeier, V., & Zimmermann, T. (2007). How documentation evolves over time. In *Proceedings of the 9th international workshop on principles of software evolution* (pp. 4–10). ACM.

Spinellis, D. (2006). *Code quality—The open source perspective*. Addison-Wesley, Pearson Education, Inc.

Tan, L., Yuan, D., Krishna, G., & Zhou, Y. (2007). /* iComment: Bugs or bad comments? */. In *Proceedings of 21st ACM SIGOPS symposium on operating systems principles* (pp. 145–158). ACM.

Tenny, T. (1988). Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering, 14*(9), 1271–1279.

Vanter, M. L. V. D. (2002). The documentary structure of source code. *Information and Software Technology, 44*(13), 767–782.

Witte, R., Zhang, Y., & Rilling, J. (2007). Empowering software maintainers with semantic web technologies. In *Proceedings of the 4th European semantic web conference* (pp. 37–52). Springer.

Yin, R. K. (2003). *Case study research—Design and methods* (3rd edn.). Sage Publications, Inc.

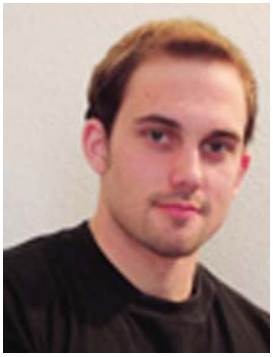
Ying, A. T. T., Wright, J. L., & Abrams, S. (2005). Source code that talks: An exploration of eclipse task comments and their implication to repository mining. In *Proceedings of the 2nd international workshop on mining software repositories* (pp. 1–5).

Zimmermann, T., Weissgerber, P., Diehl, S., & Zeller, A. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering, 31*(6), 429–445.

Author Biographies

Beat Fluri received the PhD Degree in Informatics from the University of Zurich, Switzerland, in 2008, and the MSc Degree in Computer Science from the Federal Institute of Technology (ETH), Switzerland, in 2004. He is a senior research associate in the Software Engineering Group in the Department of Informatics at the University of Zurich, Switzerland. His main research interest is software evolution, focusing on source code change analysis and recommender systems. Fluri is a member of the IEEE, the IEEE Computer Society, ACM, and the ACM SIGSOFT. More information is available at seal.ifi.uzh.ch/fluri.

Michael Würsch is a Doctoral Student and Research Assistant in the Software Engineering Group at the University of Zurich, Department of Informatics. He received his MSc Degree in Computer Science from the University of Zurich in 2007. His current research interests include software design, software evolution analysis, and service-centric software engineering. More information is available at seal.ifi.uzh.ch/wuersch.



Emanuel Giger is a Doctoral Student and Research Assistant in the Software Engineering Group at the University of Zurich, Department of Informatics. He received his MSc Degree in Computer Science from the University of Zurich in 2006. His current research interests include software design, software evolution analysis, and service-centric software engineering. More information is available at seal.ifi.uzh.ch/giger.

Harald C. Gall is professor of Software Engineering at the University of Zurich, Department of Informatics. Prior to that, he was an associate professor at the Technical University of Vienna in the Distributed Systems Group, from where he received his PhD and MSc Degree in Informatics. His research interests are in software engineering with focus on software evolution, software quality analysis, software architecture, reengineering, collaborative software engineering and service-centric software systems. Gall is also a program cochair of ICSE 2011, the International Conference on Software Engineering. Recently, he was program chair of ESEC-FSE 2005 (European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering) and MSR (International Workshop on Mining Software Repositories, co-located with ICSE) in 2006 and 2007. More information is available at seal.ifi.uzh.ch/gall.